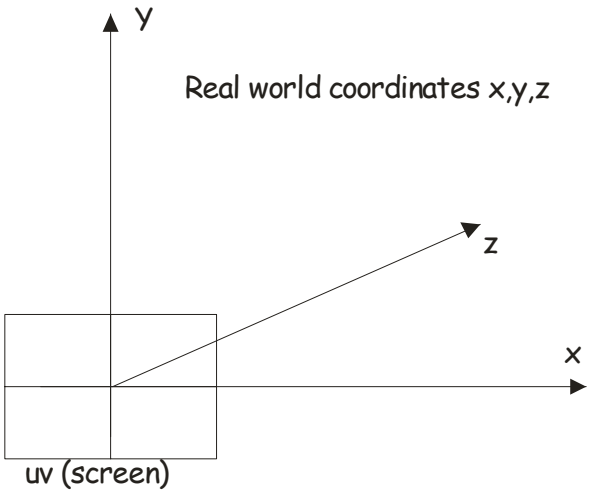
Martin
4.11.09

3D projection in milkdrop shader code

This is a text about how to write 3D shader code in milkdrop. Due to milkdrop's flat mesh, the possibilities are limited so don't expect you can turn milkdrop into a game engine.

Basic 3D projection

We see objects in the "real world", defined in a three-dimensional coordinate system (*x*,*y*,*z*), through the two-dimensional screen (*uv.x*, *uv.y*).



Let the real world x-axis point to the right, the y- axis up, and the z axis forward in viewing direction. Any point in the real world would appear on the screen at coordinates

$$uv.x = \frac{x}{z}$$

$$uv.y = \frac{y}{z}$$

This is just basic textbook projection theory, for more details see wikipedia. Actually some quite old presets used projection in the waves equations to create 3D cubes or similar.
Game engines work similarly. A typical scene in a game is made up of thousands of individual polygons, normally textured triangles, arranged in "real world" coordinates and then projected to the screen. Today's video cards do most of the job themselves, e.g. project the objects, determine hidden surfaces (you don't want to see the back of a character while he is looking at you), draw the textures etc.

Inverse projection in milkdrop shader code

How to

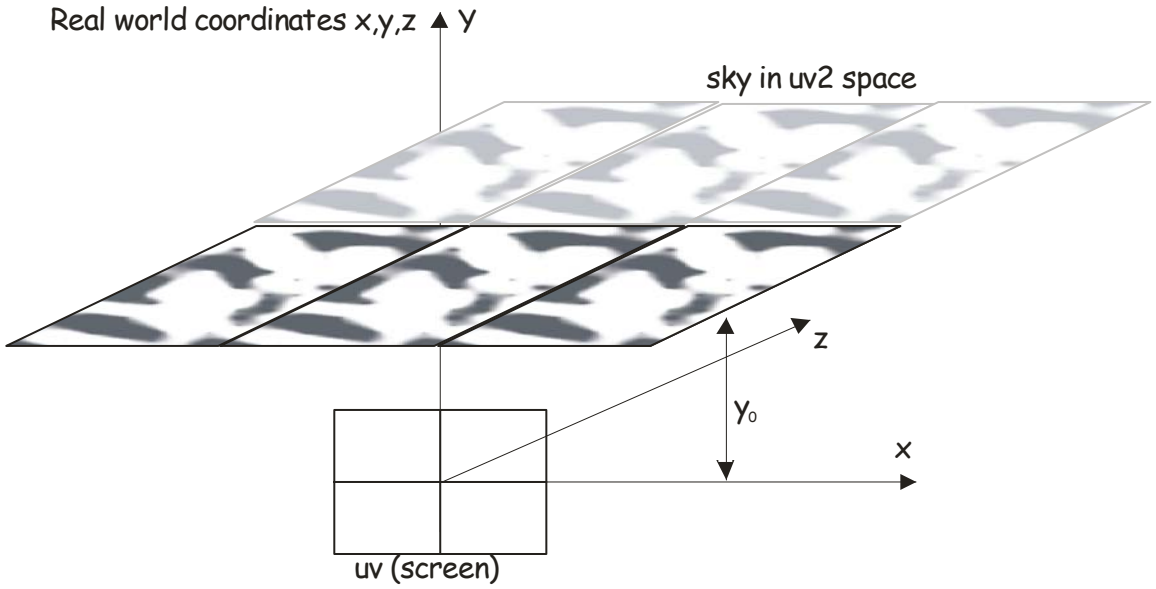
MD is not a game engine. No objects can be defined in the shader sections, and all we can work with is the screen space *uv*, ranging from (*0,0*) to (*1,1*), - in pixels e.g. (*0,0*) to (*1280,1024*). There is no z coordinate.
The video card will process the complete flat *uv* space in one go, i.e. within one frame, but it processes *only* this space and no additional objects whatsoever. The shader code is simply executed for each pixel in the *uv* space independantly, period.

All you do in the MD shader section must ultimately be based on the uv coordinates.

A simple example: We want to draw a 3D sky in the warp- or comp shader section. The sky will be painted as a noise texture so we will end up with something like

```
float sky = tex2D (sampler_noise_hq, uv2);
```

"All" we need to do is to determine *uv2* as a function of *uv*. Let's say we want our sky as a horizontal noise texture in the real world space, floating at constant height $y = y_0$, and oriented parallel to the x and z-axes as shown below. Of course, any texture is of limited size and will be tiled to +/- infinity both in x and in z direction. Looking "through" the screen, the horizon (at z = infinity) will appear in the middle.



In other words we define the relation between the real world and the sky surface *uv2* as follows

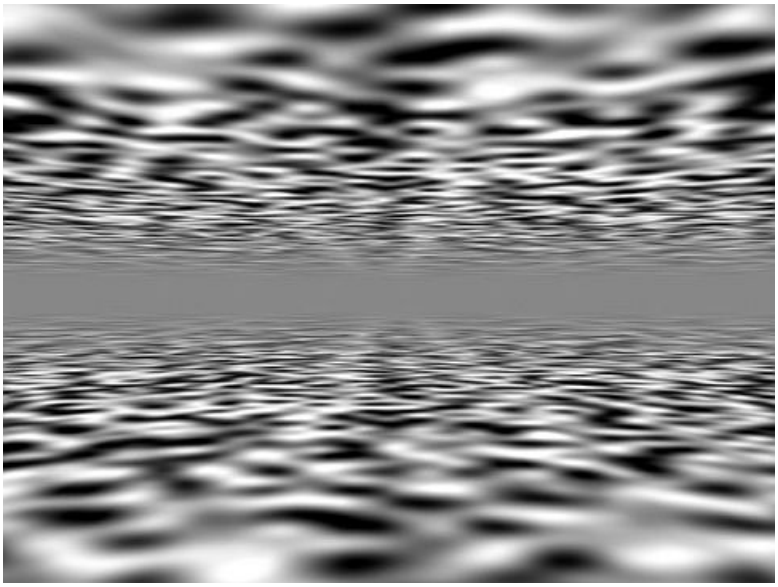
$$x = uv2.x$$
$$z = uv2.y$$
$$y = y0 = const.$$

Note that the *x,y,z* coordinates are entirely artificial and have no value in milkdrop. However we *need* them to define our surface. We will get rid of them in a second step by relating them to the screen coordinates *uv*, using the projection laws above. This gives us five conditions in total, from which we can now eliminate *x,y,z* and calculate *uv2* as a function of *uv*. Try this by yourself and you will see how the projection laws will now appear in their inverted form. That is what I call inverse projection. The result is

$$uv2.y = \frac{y_0}{uv.y}$$
$$uv2.x = uv.x \cdot \frac{y_0}{uv.y}$$

The according code could look as follows. We subtract *0.5* from *uv* first for convenience, to achieve symmetry around the screen centre rather than the upper left corner.

```
uv -= .5;
float y0 = .2;
float z = y0/uv.y; //define a variable z just for convenience
float2 uv2 = float2 (z*uv.x, z);
float sky = tex2D (sampler_noise_hq, uv2);
ret = sky;
```



You may be surprised that the sky appears mirrored as “ground”. This is a natural consequence of MD always working on the complete *uv* space and therefore texturing the whole screen. The ground is just the solution of the above equations for z > 0, while the sky represents the area where z < 0. If you don’t want this symmetry you need to actively mask out the undesired areas e.g. by multiplying the sky with a boolean condition

```
float sky = tex2D (sampler_noise_hq, uv2) * (z <= 0);
```

Basically we transformed the *screen surface uv* into a differently shaped or oriented *surface*, by a transformation *uv -> uv2*, and then *mapped*[1] a texture to this surface using tex2D ( ).
Normally in MD there is no need to talk about surfaces, because everything is by default mapped to the flat screen, i.e. the screen itself is the surface. What we have derived here is a separate surface *uv2* which is not identical to the screen but floating in an arbitrary “real world”, which itself in MD must be expressed by its relation to the *uv* space, applying the basic projection rules.

Let me reiterate: In a game engine, we would define any “object” in real space coordinates and then use the straightforward projection rules to project the object on the screen. In MD shader code, we cannot work on a “per object” basis but only on the whole screen. We need to start with the screen space *uv* and calculate backwards to obtain a texture space *uv2* which we then use for texture mapping.

This inverse projection is cumbersome, hardly intuitive (I still struggle to explain it), and very limited. It will work only for a small number of simple geometrical surfaces such as tilted planes, spheroids, tubes or cones or similar. If you start playing with more complicated surfaces, you will quickly become aware of its limitations. For instance, try to create a ripple structure for the sky, e.g. by modulating y_0 such that $y_0 = 1 + 0.1 \cdot \sin(z)$, and you will immediately end up with a system of unsolvable transcendental equations of the principal form $x = \sin(x)$. As long as no transcendent

[1] “sampling“ may be used as a synonym for mapping. In fact tex2D (sampler_noise_hq, uv2) takes “samples” of a predefined pattern at all points in the uv2 space. Since uv2 is derived from uv, it contains the same number of points, e.g. 1280 x 1024.

forms emerge, mathematical software may still find a solution for some rather simple surfaces, but chances are high that the result will go over 20 pages.

Notes

Avoiding Infinity

Since z reaches infinity in the middle of the screen, the texture will be sampled at very high frequency there, causing ugly aliasing artefacts and a high GPU load, particularly when sampling the warp shader output *sampler_main*. Therefore z should be clamped to reasonable values. The new artefacts caused by the limiter can be hidden by some form of interpolation with a solid color, creating an impression of distant fog or darkness. Example:

```
uv -= .5;
float y0 = .2;
float z = clamp (y0 / uv.y, -4, 4);
float2 uv2 = float2 (uv.x*z, z);
float sky = tex2D (sampler_noise_hq, uv2);
ret = lerp (0, sky, saturate(1/pow(z,2)) );
```

Avoiding Repetitiveness

Towards the middle of the screen, where z takes high values, the texture will be tiled to the screen in very small intervals (sampled at high frequency), and its repetitiveness may become annoyingly apparent. There is not much you can do about that if you use the warp shader output *sampler_main*, or an image as texture. For noise textures, a practical way to alleviate this problem is to sample the texture map a few times at different frequencies. Instead of

```
tex2D (sampler_noise_hq, uv2)
```

use e.g.

```
0.875 * (tex2D (sampler_noise_hq, uv2/4) + tex2D (sampler_noise_hq,
uv2/2)/2 + tex2D (sampler_noise_hq, uv2)/4)
```

or similar. This will however use up more GPU power.

Movement and rotation

We discriminate between two cases of movement:

a) related to the whole scene (the surface)
b) related to the texture mapping only

For case a), which could for instance be a rotation of the whole scene around the z-axis, **the movement equations apply to the *uv* coordinates**. If you implement your 3D code in the warp shader section, you can effectively use the built-in transformations such as *rot* to rotate the whole scene and *sy* to shift the horizon up and down. Note that MD uses different *uv* coordinates for the warp and comp shader: *uv* in the warp shader is affected by *rot, warp, zoom* etc. while *uv* in the comp shader is not. Therefore if you implement your code in the warp shader, you should make use of the MD internal transformations, they are based on the vertex mesh and faster and more efficient than in the shader directly. However if you need to implement your code in the comp shader, e.g. because you want to use the warp shader output as texture, you will have to write your own code such as

```
uv -= -5.:
float angle = 0.3;
```

```
uv = mul (uv, float2x2(cos(angle), sin(angle), -sin(angle), cos(angle));
etc.
```

Note that the sines and cosines need only be calculated once per frame so please do not put them into the shader code but move them to the frames equations section. The mul operation needs to remain in the shader of course because it will be applied to each pixel individually.

Case b), movement of texture only, leaves the surface alone and only shifts the texture along the surface. This can be used e.g. to create an illusion of moving forward or sidewards within the scene, and is **applied to *uv2* only.** For a movement towards the screen, use e.g.

```
float2 uv2 = float2 (uv.x*z, z + time);
```

Overcoming the “flat” appearance

Regardless which surface you define to map your texture to, there will always be a general impression of flatness. The sky does not appear like clouds but rather like a slide projected onto a flat surface, because, well, it *is* just a pattern wallpapered to a surface.

To overcome this, the texture should be correlated to the surface: Dark areas in the sky should appear at another height y_0 than bright areas, same as in the real world. If you look over a field, you’ll see lumps of soil and stones of different colours, texturing the field *and* forming its surface. This is what we try next.

Generating profiled 3D surfaces

As said above, the inverse projection problem can only be solved for trivial surfaces, described by a function in real world coordinates which is at least invertible. Invertibility is necessary but not sufficient for a closed solution, as seen for the sine function, which *is* invertible, but still leads to unsolvable equations.

When we now try to make the surface dependant of its texture, we have the problem that a noise texture is not invertible in the first place.
A texture is defined by *color = function (uv) = tex2D (sampler_noise_hq, uv)*, while the inverse would be $uv = function^{-1} (color)$ which apparently has no unique solution. Rather there will be many points in the *uv* space for which the texture will yield a given color. Nevertheless there is a trick to bypass this problem.

Resuming the sky example above, to make the clouds more realistic, we want the height of the sky be modulated with the noise texture. It would sound reasonable to simply add the sky texture to the height y_0. However we need the sky texture sampled in *uv2* space before we can add it to y_0. But we don’t have it, we must first calculate *uv2*, which itself is based on y_0. The trick to break the circle is to take an iterative approach: calculate the sky as above with constant y_0, then add (a small fraction of) the resulting sky texture to y_0, then go back and recalculate the sky with the new y_0 etc.

In the sample code below, I start with the code above but append a loop which does four iterations. Two to four iterations should normally be sufficient. Do not exaggerate as too much iteration will slow your code down.

The variable *depth* determines the fraction of sky texture to be added to y_0. The value should be quite small; otherwise you will get ugly distortions, simply because the whole approach *is* only an approximation and not an exact solution of the inverse projection.

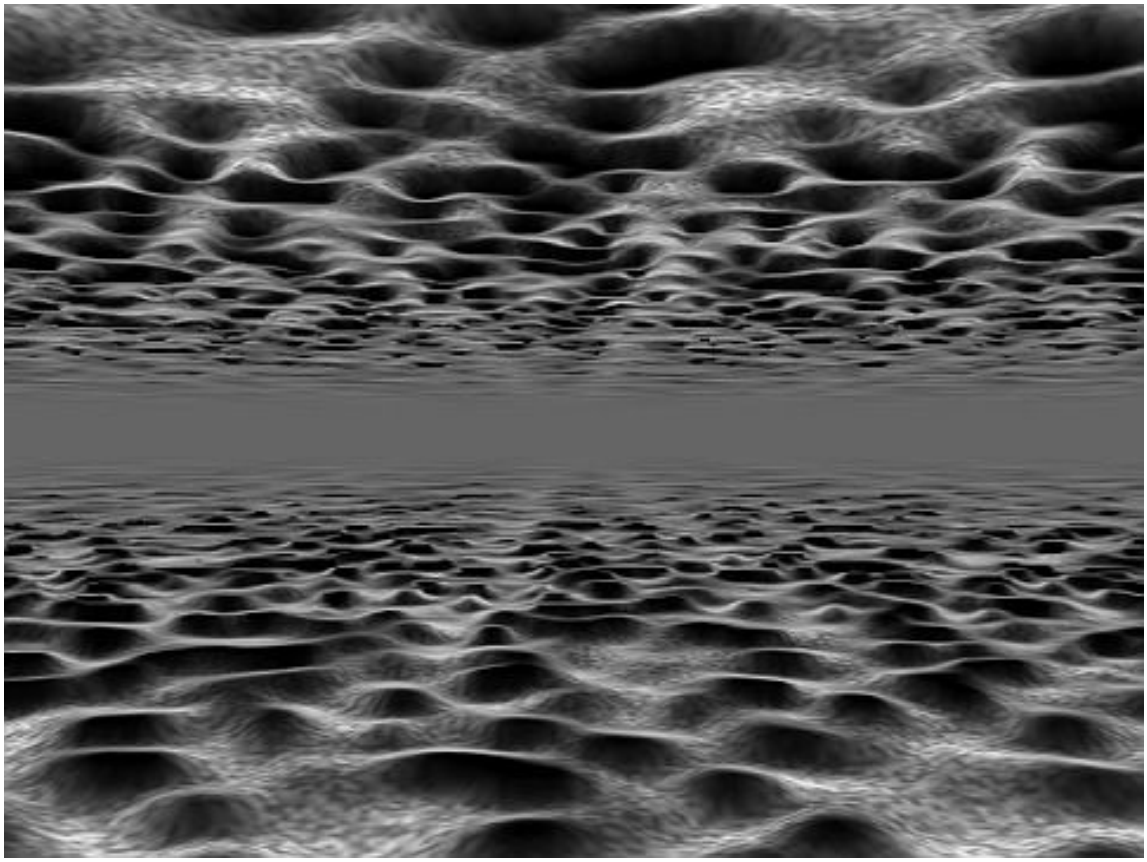
It is a good idea to sprinkle a bit of low quality, high frequency noise such as *sampler_noise_lq* in the end, to improve the impression of the structured profile.

```
uv -= .5;
float y0 = .2;
float z = y0/uv.y;
float2 uv2 = float2 (z*uv.x, z);
float sky = tex2D (sampler_noise_hq, uv2);

float depth = .006;
for (int n = 1; n <= 4; n++)
{
  y0 += depth*sky; //alternatively y0 *= 1+depth*sky;
  z = y0/uv.y;
  uv2 = float2 (z*uv.x, z);
  sky = tex2D (sampler_noise_hq, uv2);
}

sky *= 1-.5*tex2D (sampler_noise_lq,uv2);
ret = sky;
```

And here is the result. The sky does not really look like clouds as originally intended but that is just a matter of colouring and fine tuning.



The code may require some refinement in practice, e.g. you may find it appropriate to scale *depth* with z to avoid excessive distortions towards the horizon.

The same scheme can of course be equally applied to other surfaces / textures. Interesting effects can be achieved using the warp shader output *sampler_main*, or non-smoothed noise textures such as *sampler_pw_noise_lq*.